Linpeng Zhang
Matricola: 1162114
3 febbraio 2019

Beauty

Relazione sul progetto di Programmazione ad oggetti

Titolo: Beauty

Tempo impiegato: circa 50 ore, di cui

- Analisi del problema: 2 h
- Progettazione modello: 3 h
- Progettazione GUI: 3 h
- Apprendimento libreria Qt: 6 h
- Codifica modello: 12 h
- Codifica GUI: 10 h
- Debugging: 2h
- Testing: 2h
- Tutorato: 10h

Compilazione ed esecuzione:

Il progetto prevede l'utilizzo di un file Beauty.pro diverso da quello ottenibile con l'esecuzione qmake -project, a causa della presenza di funzionalità di c++11, come per esempio lambda-espressioni e keywords come default ed override.

Inoltre è stato aggiunto un file data.xml che può (eventualmente) essere aperto dall'applicazione e che contiene alcuni dati di test pronti all'uso.

L'intero progetto è stato realizzato su sistema operativo macOS Mojave 10.14.3, Qt creator 5.12 e clang-1000.10.44.4.

Descrizione

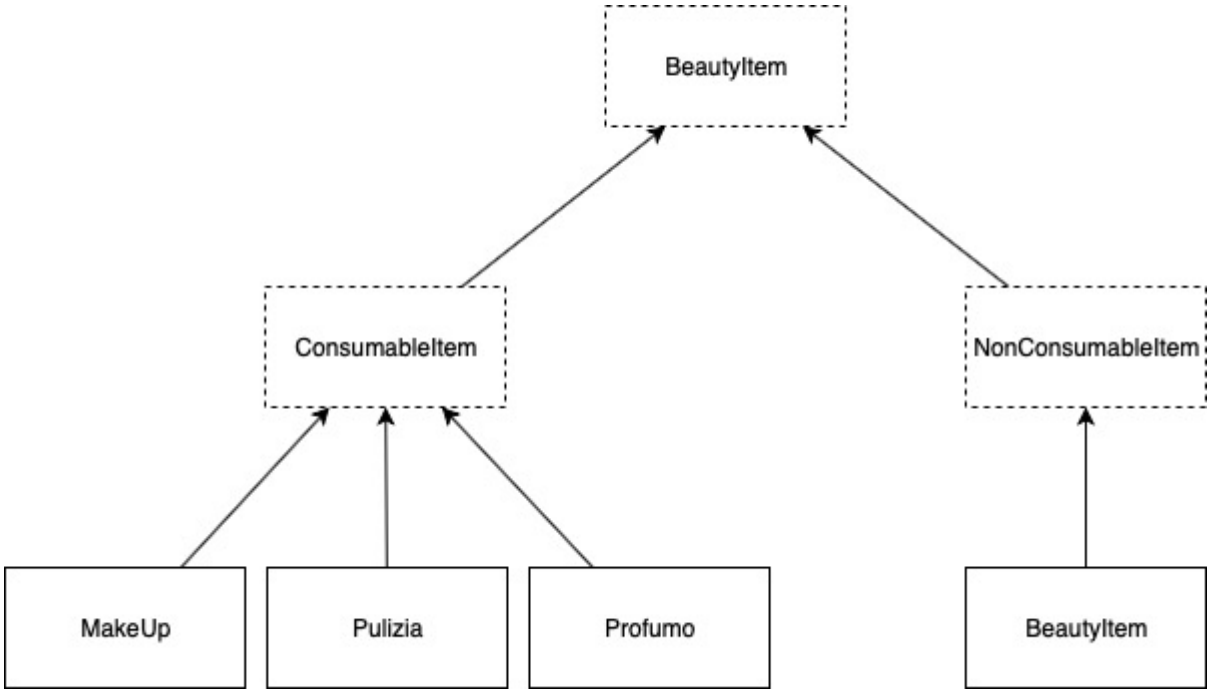
Beauty è un applicazione che consente di gestire contenitori di oggetti che solitamente si trovano in un beauty: per esempio trucchi, profumi, saponi, strumenti vari.

Beauty consente di creare nuovi file, aprirne di già esistenti ed ovviamente salvarli. Ogni file rappresenta un contenitore su cui è permesso l'inserimento, la rimozione e la ricerca di oggetti. Oltre a ciò, Beauty permette di indicare le quantità disponibili di ciascuno, o il numero di volte che si è utilizzato un certo oggetto, e gestirne l'utilizzo.

Progettazione

È stato deciso di aderire al design pattern Model-View, perché Qt fornisce una serie di classi “View” che sfruttano questa architettura Model-View che consente una separazione tra il modello logico e la GUI. La flessibilità di questo pattern, grazie agli strumenti forniti da Qt, e il monte ore superiormente limitato sono i principali motivi per cui è stata adottato tale pattern, nonostante la consapevolezza della possibilità offerta dal pattern Model-View-Controller di separare anche la View dal Controller.

Gerarchia di tipi



La classe base astratta polimorfa BeautyItem rappresenta la classe base della gerarchia e rappresenta un qualsiasi oggetto astratto che può essere messo in un beauty. Rappresenta un oggetto astratto perché non si riferisce a una specifica tipologia di oggetti; possiede infatti metodi virtuali puri che verranno introdotti successivamente.

La classe BeautyItem presenta due campi dati privati: marca e nome, entrambi di tipo stringa, che rappresentano proprietà che crediamo che qualsiasi oggetto di un beauty abbia.

Inoltre BeautyItem possiede un campo dati **protetto statico** static std::map<std::string, BeautyItem*>. Questo permette di associazione una stringa, che rappresenta il tipo di una classe derivata D, ad un puntatore polimorfo p con TS(p) = BeautyItem* e TD(D) = D*. Questa mappa, chiaramente protetta perché utilizzata esclusivamente dalle classi derivate, viene sfruttata per la serializzazione, di cui accenneremo nella sezione: *Serializzazione*. Si noti che ogni sottoclasse concreta della gerarchia possiede un campo dati privato **statico** che modifica questa mappa.

I sottotipi diretti della classe BeautyItem sono:

- ConsumableItem: è una classe polimorfa astratta e rappresenta oggetti di un beauty che hanno una capacità e si esauriscono, come, per esempio, profumi, saponi, trucchi spalmabili. ConsumableItem è dotato di due campi dati privati: float capacità e float consumo per gestirne gli utilizzi.

- NonConsumableItem: è una classe polimorfa astratta e rappresenta oggetti di un beauty che non hanno una capacità, come, per esempio, pettini e spazzolini. NonConsumableItem è dotato di due campi dati privati: un unsigned int contatore che conta gli utilizzi effettivi e un campo materiale di tipo stringa.

ConsumableItem ha tre sottotipi concreti polimorfi:

- MakeUp: rappresenta i trucchi che si possono inserire nel beauty. È dotato di un campo dato privato colore di tipo stringa.
- Pulizia: rappresenta tutti i prodotti utilizzabili per la pulizia e per la cura della pelle, come per esempio saponi e gel detergenti. È dotato di un campo dato privato parteDelCorpo di tipo stringa che contiene dove si utilizza il prodotto in questione.
- Profumo: rappresenta i profumi. È dotato di un campo dato privato std::string odore.

NonConsumableItem ha un sottotipo concreto polimorfo:

- Strumento: rappresenta strumenti che vengono generalmente inseriti in un beauty, i quali pettini, spazzolini o altro. Contiene un campo dati privato std::string parteDelCorpo che indica dove si utilizza tale strumento.

Si noti che ogni classe di questa gerarchia fornisce metodi pubblici get/set per i propri campi dati privati.

Metodi Polimorfi

La classe BeautyItem ha i seguenti metodi virtuali puri **pubblici**, di cui omettiamo quelli per la serializzazione per discuterne nel paragrafo seguente:

- virtual std::string tipo() const = 0: ogni sua implementazione deve ritornare una stringa che identifica la classe derivata; può essere utilizzato sia per la serializzazione, di cui parleremo successivamente, sia per avere indicazioni sul tipo di un oggetto polimorfo senza ricorrere al dynamic_cast (che avviene attraverso RTTI), ma con il dynamic binding che ha un overhead minore in quanto si sfruttano le vtables.
- virtual bool use() = 0: è un metodo che permette di gestire l'utilizzo di un oggetto; nella sottoclassie astratte ConsumableItem è stato implementato in modo che venga ridotta la capacità rimanente; nella sottoclasse astratta NonConsumableItem, invece, viene incrementato il contatore degli utilizzi. Nulla vieta che ci possano essere ulteriori override, magari in presenza di nuove classi derivate e favorendo in generale l'estensibilità della gerarchia.
- virtual bool isConsumable() const = 0: è un metodo che permette di capire se un determinato oggetto polimorfo sia un sottotipo di ConsumableItem. È stato implementato come "final override" nelle sottoclassi ConsumableItem e NonConsumableItem: infatti un sottotipo di ConsumableItem è certamente un ConsumableItem e non avrebbe senso consentire ulteriori override.
- virtual BeautyItem* clone() const = 0: è il metodo di clonazione polimorfa e restituisce un puntatore ad una copia dell'oggetto d'invocazione. Si noti che è stato implementato solamente nelle sottoclassi concrete, poiché non si possono stanziare classi astratte.

Naturalmente BeautyItem ha il distruttore virtuale **pubblico**:

- virtual ~BeautyItem() = default: permette l'invocazione del giusto distruttore da parte di qualsiasi oggetto polimorfo.

Serializzazione

Beauty consente il salvataggio e l'apertura di file che rappresentano i contenitori. Per fare ciò utilizza file XML che hanno il vantaggio di essere facilmente comprensibili dalle persone, oltre ad avere numerosi strumenti disponibili per la loro manipolazione. Inoltre Qt fornisce diverse classi che ne facilitano notevolmente l'utilizzo, nascondendone gli aspetti implementativi.

Innanzitutto la classe BeautyItem rappresenta i seguenti metodi virtuali puri:

- virtual BeautyItem* create(QXmlStreamReader&) const = 0: è un metodo virtuale puro **privato**. Ogni sottoclasse concreta D che, ovviamente, implementa questo metodo, deve leggere da uno stream QXmlStreamReader gli opportuni campi, creare un nuovo oggetto della classe derivata e ritornare un puntatore a tale oggetto; verrà quindi ritornato un D* che va bene per covarianza sul tipo di ritorno.
- virtual void serialize(QXmlStreamWriter&) const = 0: è un metodo virtuale puro pubblico. Ogni sua implementazione deve scrivere correttamente i propri campi dato su uno stream di tipo QXmlStreamWriter, che verranno eventualmente letti dal metodo precedente per creare il giusto oggetto. Chiaramente, oltre ai campi dati, verrà scritto anche un'informazione riguardante al tipo dell'oggetto in questione, per esempio sfruttando il metodo virtuale puro std::string tipo().

Oltre a ciò è necessario un metodo statico nella classe base BeautyItem che effettui la lettura del file chiamando il giusto metodo create, motivo per cui l'indicazione sul tipo dell'oggetto da creare va inserito all'inizio della scrittura del file. Per fare ciò è stato aggiunto un campo dati protetto statico nella classe base BeautyItem: static std::map<std::string,BeautyItem*> table. Questa mappa associa una stringa che contiene indicazioni sul tipo D di un oggetto ad un puntatore polimorfo che dinamicamente punta a D. Possiamo allora implementare il seguente metodo nella classe base:

- static BeautyItem* unserialize(QXmlStreamReader&) const: è un metodo statico pubblico (non virtuale) che legge il tipo dal QXmlStreamReader, controlla se nella mappa c'è il tipo in questione ed in caso positivo chiama il metodo create. La chiamata sarà del tipo: *table[tipo]->create(streamreader)*, dove il late binding ci permette di selezionare il metodo create corretto a run time.

Per quanto riguarda l'inizializzazione della mappa, è stata inserita in ogni classe concreta derivata D un campo dato statico privato di tipo classe StaticTableInit. Questa classe, annidata nella parte privata di D, ha un costruttore che costruisce un oggetto di tipo D e aggiunge un puntatore a tale oggetto nella tabella. Inoltre questo puntatore viene memorizzato come campo dato di StaticTableInit: ciò consente di distruggere, una volta terminato il programma, l'oggetto creato ridefinendo ~StaticTableInit(). Il codice del costruttore sarà di questo tipo:

```
ClasseDerivata::StaticTableInit::StaticTableInit(){
    ptr = new ClasseDerivata();
    table[ptr->tipo()]=ptr;
}
```

Il frammento di un file XML che ne risulterà da un BeautyItem sarà di questo tipo:

```
<BeautyItem tipo="Indicazione_sul_tipo (per esempio: oggetto->tipo())">
<campodato1>dato</campodato1>
```

…

<campodaton>dato</campodaton>

</BeautyItem>

dove ovviamente il modello cliente della gerarchia può avere tanti BeautyItem (per esempio tutti quelli di un singolo contenitore) salvati in un unico file.

Container<T>

Il modello logico dei dati sfrutta un opportuno contenitore. Poiché è essenziale l'accesso in posizione arbitraria in tempo costante O(1) è stato deciso di implementare un array dinamico. Ciò consente anche l'inserimento e la rimozione in coda in tempo ammortizzato costante.

Tuttavia, l'applicazione consente anche la rimozione in posizione arbitraria. Certamente una lista concatenata è più efficiente in questo caso (infatti consente inserimento e rimozione in posizione arbitraria in tempo O(1), contro il tempo ammortizzato O(n) dell'array), ma le motivazioni, in ordine di importanza decrescente, che ci spingono a preferire un array dinamico sono le seguenti:

- gli accessi in posizione arbitraria sono molto più delle rimozioni. Ogni volta che si manipola la View, per esempio per visualizzare, modificare o usare un campo dato o un determinato oggetto, si accede ad un elemento in posizione arbitraria;
- per rimuovere un elemento in posizione arbitraria potremmo non avere a disposizione l'iteratore corrispondente a tale elemento, ma solamente la posizione intera. In questo caso, che corrisponde al modello logico di Beauty, il vantaggio della rimozione in posizione arbitraria in tempo O(1) delle liste è solamente apparente: includendo la ricerca arriva a O(n);
- il contenitore può contenere un numero di oggetti arbitrario, ma dalla nostra analisi dei requisiti crediamo che questo numero non sia troppo elevato. Per esempio, un contenitore di 1000 puntatori a BeautyItem occupa circa 8000 B, che possono essere inseriti in memoria cache, garantendo buone prestazioni.

Si noti che è stato deciso di implementare un template di classe DeepPtr<T> che opera in modalità profonda. Il contenitore, pertanto, conterrà oggetti di tipo DeepPtr<T>, che contengono un campo dati privato di tipo BeautyItem* (naturalmente polimorfo).

È stata anche implementata una classe XmlIO per gestire la serializzazione del modello dati, e che sfrutta i metodi forniti dalla gerarchia di classi.

Libreria Qt

È stato deciso di sfruttare come View la QTableView fornita da Qt. Esso si appoggia su un delegate predefinito e un modello custom QFilterProxyModel che è sottotipo di QSortFilterProxyModel (fornito da Qt). Questo modello fa da ponte con il modello custom QTableModelAdapter che deriva dalla classe astratta QAbstractTableModel: in particolare consente la ricerca di oggetti nel Beauty, che verranno visualizzate correttamente nella QTableView; allo stesso modo, una modifica sulla view deve propagarsi nella giusta cella del QTableModelAdapter, e ciò è facilitato dai framework forniti da Qt.

QTableModelAdapter implementa tutti i metodi per la corretta visualizzazione dei dati e per la modifica dei dati. Per garantire incapsulamento, modularità e manutenibilità, esso è separato dal vero modello logico dei dati, ma contiene solamente un puntatore a quest'ultimo.